Concepts et Model Checking

Stefan Schwoon

ENSIIE, Année 2024/2025

Organisation

Cours: lundi de 14h à 15h45

TD/TP: lundi de 16h à 17h45

Enseignant: Stefan Schwoon (`schwoon@lmf.cnrs.fr`)
`http://www.lsv.fr/~schwoon`

Durée: 6 semaines (du 20 janvier au 3 mars)

Évaluation: Examen (le 3 mars)

Connaissances préalables:

logique, théorie d'automates . . .

Littérature:

Clarke, Grumberg, Peled: Model Checking, MIT Press, 1999

Baier, Katoen: Principles of Model Checking, 2008

Emerson: Temporal and Modal Logic, chapître 16 du *Handbook of Theoretical Computer Science*, vol. B, Elsevier, 1991

Vardi: An Automata-Theoretic Approach to Linear Temporal Logic, LNCS 1043, 1996

Holzmann: The SPIN Model Checker, Addison-Wesley, 2003

Partie 1: Introduction

Que veut dire “Model-Checking” ?

Notion de logique :

Tester si un object (p.ex. affectation des variables) est modèle d’une formule.

Ici : appliquer de la logique pour vérifier la correction d’un système

Logique temporelle : extension de la logique du premier ordre

Rappels: Calcul/Logique propositionnel(le)

Formules avec prédicats :

$A \quad \widehat{=} \quad$ “Anne est architecte”

$B \quad \widehat{=} \quad$ “Bruno is boulanger”

and connecteurs, p.ex. $\wedge$ (“et”), $\vee$ (“ou”), $\neg$ (“non”), $\rightarrow$ (“implique”).

Exemples

Formules de logique propositionnelle :

$A \wedge B$ (“Anne est architecte et Bruno est boulanger”)

$\neg B$ (“Bruno n’est pas un boulanger”)

Une telle formule, est-elle vraie ?

Ça depend.

Certaines formules sont toujours vraies ($A \vee \neg A$) ou toujours fausses ($B \wedge \neg B$).

Mais en général, les formules sont évaluées par rapport à une affectation des prédicats.

Logique propsitionnelle

Une affectation $\mathcal{B}$ est une fonction qui donne (1 ou 0) pour tout prédicat.

La semantique d'une formule (définie inductivement) est l'ensemble des affactions qui rendent la formule vraie, notée $[\![F]\!]$. P.ex.,

Si $F = A$ alors $[\![F]\!] = \{ \mathcal{B} \mid \mathcal{B}(A) = 1 \}$;

Si $F = F_1 \wedge F_2$ alors $[\![F]\!] = [\![F_1]\!] \cap [\![F_2]\!]$; ...

D'autres notations: $\mathcal{B} \models F$ ssi $\mathcal{B} \in [\![F]\!]$.
On dit : "$\mathcal{B}$ satisfait F" ou "$\mathcal{B}$ est un modèle de F".

Model-checking pour logique propositionnelle

Problème:

Étant donné une affectation $\mathcal{B}$ et une formule F du calcul propositionnel, tester si $\mathcal{B}$ est un modele de F.

Solution:

Remplacer les prédicats par leurs valeurs dans $\mathcal{B}$, utiliser le tableau de vérité pour voir si ça donne 1 ou 0.

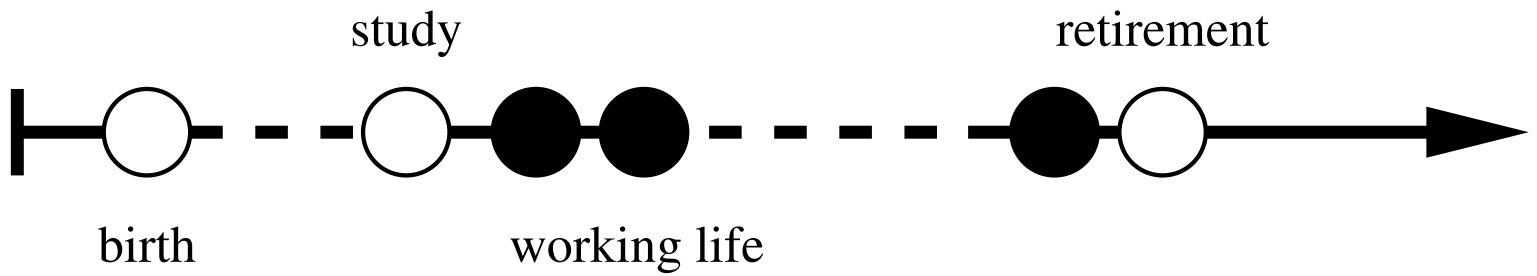
Exemples: Let $\mathcal{B}_1(A) = 1$ et $\mathcal{B}_1(B) = 0$. Alors $\mathcal{B}_1 \not\models A \wedge B$ et $\mathcal{B}_1 \models \neg B$.

Let $\mathcal{B}_2(A) = 1$ et $\mathcal{B}_2(B) = 1$. Alors $\mathcal{B}_2 \models A \wedge B$ et $\mathcal{B}_2 \not\models \neg B$.

Logique temporelle

On permet aux prédicats de changer au fil de temps.

Exemple: Valeurs de A dans la vie d'Anne:



Propriétés qu'on veut exprimer :

Anne sera finalement architecte (à un point dans le futur).

Anne sera architecte jusqu'à ce qu'elle prend sa retraite.

$\Longrightarrow$ Extension de la logique propositionnelle avec des modalités temporelles ("finalement", "jusqu'à").

Aperçu

Logique temporelles linéaire (exemple: LTL)

formules avec modalités temporelles

évaluée sur les *séquences* (infinies) d'affectations

Le problème de model-checking pour LTL: Étant donné une formule de LTL et une séquence de valuations, tester si la séquence est un modèle de la formule.

Logique temporelles branchantes (CTL, CTL*)

On considère des *arbres* (infinis) d'affectations

Interprétation: nondéterminisme; plusieurs développement potentiels

Le rapport avec la vérification de programmes

Espace d'états d'un programme:

- compteur d'instruction
- valeurs de variables
- contenu de la pile, du tas, ...

Prédicats, p.ex.

- "x possède une valeur positive."
- "Le compteur d'instructions est dans la ligne ℓ."

Étant donné un ensemble de prédicats, un état du programme donne une affectation de ces prédicats.

Programmes et logique temporelle

Logique temporelle linéaire:

Toute exécution donne une séquence d'affectations.

Interprétation du programme: l'ensemble de ces séquences

Question : La formule, est-elle satisfaite par toutes les séquences ?

Logique branchante:

Le programme peut brancher à certains endroits, ses exécutions forment un arbre d'affectations.

Interprétation du programme: arbre avec l'état initial comme racine

Question : Cet arbre, satisfait-il la formule ?

Donc: problème de vérification $\widehat{=}$ problème de model-checking

Exemple: Quicksort - correct ou non ?

```
void quicksort (int left, int right) {
   int lo,hi,piv;
   if (left >= right) return;
   piv = a[right]; lo = left; hi = right;
   while (lo <= hi) {
     if (a[hi] > piv) {
       hi = hi - 1;
     } else {
       swap a[lo],a[hi];
       lo = lo + 1;
     }
   }
   quicksort(left,hi);
   quicksort(lo,right);
}
```

Exemple : Quicksort

On considère que l'algorithme est correct si :

Il trie correctement.

Il termine pour tout argument légal.

Dans ce cas, l'algorithme ne termine pas toujours,
notamment quand `a[right]` est l'élément maximal.

Remarque : Ce bug aurait peut être trouvé par testing rigureux.

Bug du processeur Pentium (1994)

Le Pentium produisait de faux résultats pour certains opérations mathématiques:

$$4195835 - (4195835/3145727) \times 3145727 = 256$$

La raison en étaient quelques valeurs fausses dans un tableau précalculé pour effectuer la division.

Coût approximatif pour Intel : 500 millions de dollars

Exemple: Variables partagées

Démonstration : counter.c

On crée deux threads qui augmentent une variable partagée n fois.

Résultat attendu : $2n$

Exemple: Variables partagées

Démonstration : counter.c

On crée deux threads qui augmentent une variable partagée n fois.

Résultat attendu : $2n$

Pourtant, le résultat réel est souvent moins que $2n$.

$\rightarrow$ Condition de compétition (*race condition*),
facile à manquer lors d'une inspection manuelle du code.

$\rightarrow$ Solution : Assurer *exclusion mutuelle* sur l'accès à n.

Exemple: Exclusion mutuelle (Peterson)

Variables partagées : `flag[0], flag[1], victime`, initialement 0

Code du processus i=0,1 (autour d'une zone critique) :

```
while (1) {
  ...
  autre = 1-i;
  flag[i] = 1;
  victime = i;
  while (victime == i && flag[autre]);
  // zone critique
  flag[i] = 0;
  ...
}
```

Exemple: Exclusion mutuelle (Peterson)

Variables partagées : `flag[0], flag[1], victime`, initialement 0

Code du processus i=0,1 (autour d'une zone critique) :

```
while (1) {
  ...
  autre = 1-i;
  flag[i] = 1;
  victime = i;
  while (victime == i && flag[autre]);
  // zone critique
  flag[i] = 0;
  ...
}
```

L'algorithme assure bien l'exclusion mutuelle. Pourtant, sa correction est déjoué par les optimisations faites sur les processeur modernes (réordonanncements des read et write).

Approches pour assurer la correction des systèmes

Éviter les erreurs:

- langages de programmation appropriés
- méthodes du génie logiciel

Détecter les erreurs:

- Simulation, testing

Prouver leur absence:

- Vérification déductive (Hoare, preuve automatique)
- Vérification automatique (model checking)

Simulation et testing

Trouver des erreurs dans la phase de conception (simulation) ou dans le produit final (testing).

Méthodes: Blackbox/whitebox testing, critères de couverture, etc

Avantages: peut trouver des erreurs rapidement et économiquement

Inconvénients: incomplet

$\rightarrow$ Aucun critère de couverture ne garantit l'absence des erreurs.

$\rightarrow$ Pas du tout adapté aux effets non-déterministes tel que la concurrence.

Testing et vérification

Simulation et testing peuvent identifier des erreurs mais pas prouver leur absence. Ces méthodes considèrent un sous-ensemble des exécutions potentielles.

$\rightarrow$ pas suffisant pour des aspects de sécurité

La vérification considère toutes les exécutions d'un système

$\rightarrow$ on peut prouver l'absence d'erreurs
(mais c'est plus couteux/difficile à mettre en œuvre)

Vérification déductive

Preuve par sémantique formelle du programme (Dijkstra, Hoare et al.)

Exemple: Logique de Hoare:

$$\{P\}\ S\ \{Q\}$$

“Si P est vrai avant l’exécution de S, alors Q est vrai après.”

Règles de preuve, p.ex.:

$$\{P\}\ \mathbf{skip}\ \{P\} \qquad \{P[x/e]\}\ x := e\ \{P\} \qquad \frac{\{P\}\ S_1\ \{Q\} \wedge \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

Exemple: régle de preuve pour les boucles

$$\{P\}\ \mathbf{while}\ \beta\ \mathbf{do}\ C\ \{Q\}$$

Il faut trouver une invariante I avec les propriétés suivantes :

$$P \Rightarrow I \qquad \{I \wedge \beta\}\ C\ \{I\} \qquad I \wedge \neg\beta \Rightarrow Q$$

Terminaison: trouver une fonction $f(x, y, \ldots)$ sur les variables telle que

$$\{\beta \wedge f(x, y, \ldots) = k\}\ C\ \{f(x, y, \ldots) < k\} \qquad f(x, y, \ldots) \leq 0 \Rightarrow \neg\beta$$

Le programme C est considéré correct si $\{true\}\ C\ \{P\}$, où P est la propriété finale désirée.

Vérification déductive

Avantages:

Complète; limitée seulement par l'ingéniosité humaine.

Inconvénients:

voir ci-dessus

Preuves manuelles lourdes (peut-être aidé par la démonstration automatique).

Le schéma ci-dessus marche pour les systèmes séquentielles (pas de concurrence).

Plutôt conçu pour les programmes genre entrée/sortie, mais pas pour les systèmes réactives.

Systèmes réactives

Exemples: système d'exploitation, serveurs, distributeur de billets, ...

pas de "fonction" calculée, terminaison non-désirable

On s'intéresse à certains propriétés de leur exécutions telles que :

Absence de blocage

Exclusion mutuelle dans une "zone critique"

Progrès: un processus qui souhaite entrer dans une zone critique y parviendra finalement.

$\Rightarrow$ formalisation par logique temporelle

Model checking

Généralement parlant, le terme model checking est donné aux méthodes qui :

verifient automatiquement si un système satisfait une spécification donnée;

soit prouvent la correction du système par rapport à la spécification;

soit trouvent un contre-exemple, une exécution qui ne respecte pas la spécification (au moins dans le cadre de LTL).

Les “pros et cons” du model checking

Avantages:

automatique(!)

bien adapté aux systèmes réactifs, concurrents, distribués

on peut tester des propriétés complexes, pas just l’accessibilité

Inconvénients:

En général, les programmes sont aussi expressifs qu’une machine de Turing
$\rightarrow$ indécidabilité

Approche: on étudie des sous-classes où le problème reste décidable, p.ex. les automates finis

Espace d’états souvent très, très large $\rightarrow$ (algorithmiquement) couteux

approche: algorithmes et structures de données efficaces

Limites du model checking

On ne peut pas espérer de vérifier n'importe quelle propriété de n'importe quel programme !

Il faut éventuellement considérer un modèle simplifié d'un programme focalisé sur ses aspects "importants".

La construction d'un tel modèle, la spécification et la vérification elle-même sont coûteux et nécessitent un effort.

$\Rightarrow$ utile dans les premières phases de conception

$\Rightarrow$ indispensable lorsque les erreurs sont très coûteuses ou même fatales (processeur, protocôlesi de communication, avions, . . .)

Succès du model checking

Depuis les années 1970: recherche sur les fondations théoriques

Depuis les années 1990: applications dans l'industrie

D'abord vérification de matériel, puis logiciel :

- vérification du protocôle de cohérence de cache dans le IEEE Futurebus+ (1992)

  L'outil SMV était en mésure de trouver plusieurs bugs quatre ans après la conception initial du bus.

- vérification de l'unité arithmétique du Pentium4 (2001)
- Static Driver Verifier (Microsoft, 2000–2004) (pilotes dans Windows)

groupes de recherche dans les grandes entreprises: IBM, Intel, Microsoft, ...

Prix Turing 2007 pour les pionniers (Clarke, Emerson, Sifakis)

Objectifs du cours

Fondements du model checking, théorie, applications

Modélisation: systèmes de transition, structures de Kripke; outils: Spin, SMV

Spécification: LTL, CTL

Vérification: techniques fondamentales et extensions (BDDs, abstraction)

Partie 2: Structures de Kripke

Modèles

On étudie un modèle très générique, les systèmes de transitions (ST):

$$\mathcal{T} = (S, \rightarrow, r)$$

S	$\widehat{=}$	espace d'états; les états que le système peut atteindre (ensemble fini ou infini)
$\rightarrow \subseteq S \times S$	$\widehat{=}$	relation de transitions; décrit les actions possibles
$r \in S$	$\widehat{=}$	état initial ("racine")

Exemple 1: Producteur/Consommateur

Pseudocode avec variables et concurrence:

var *turn* {0,1} **init** 0;
cobegin {*P* ‖ *K*} **coend**

P =

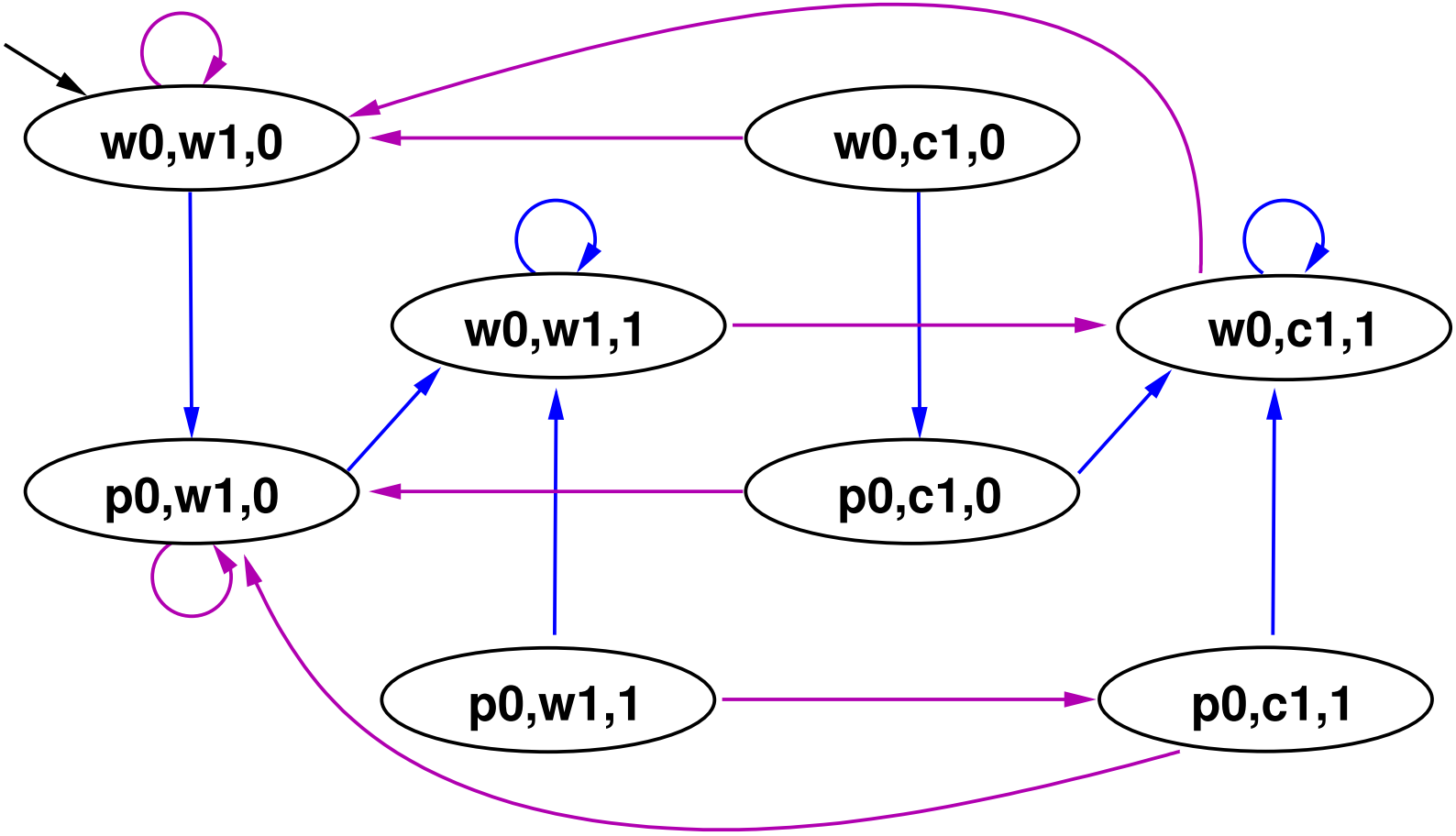
start;
while true do
  w_0: **wait** $(turn = 0)$;
  p_0: /* produce */
    $turn := 1$;
od;
end

K =

start;
while true do
  w_1: **wait** $(turn = 1)$;
  c_1: /* consume */
    $turn := 0$;
od;
end

Exemple 1: ST correspondant

$S = \{w_0, p_0\} \times \{w_1, c_1\} \times \{0, 1\}$; racine $(w_0, w_1, 0)$



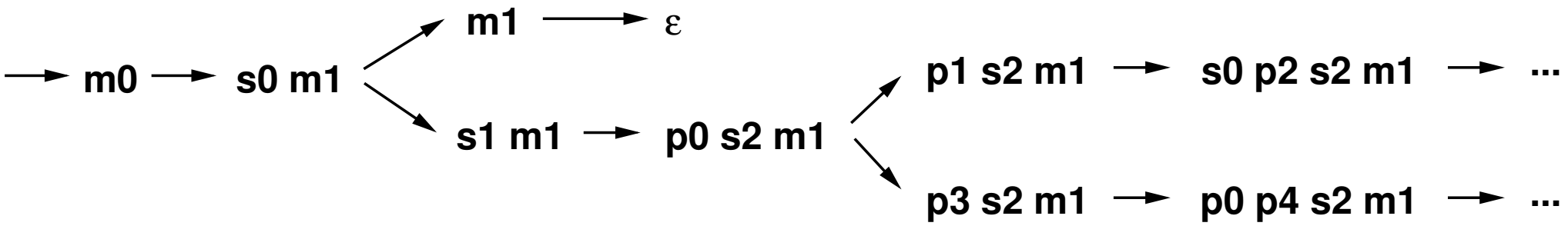
Exemple 2: Programme recursif

procedure p;
p_0: **if** ? **then**
p_1: **call** s;
p_2: **if** ? **then call** p; **end if**;
else
p_3: **call** p;
end if
p_4: **return**

procedure s;
s_0: **if** ? **then return**; **end if**;
s_1: **call** p;
s_2: **return**;

procedure $main$;
m_0: **call** s;
m_1: **return**;

$S = \{p_0, \ldots, p_4, s_0, \ldots, s_2, m_0, m_1\}^*$, racine m_0



Notations pour ST

On écrit $s \to t$ si $(s, t) \in \to$.

Si $s \to t$ alors s s'appelle prédecesseur direct de t et t successeur direct de s.

S^* dénote les séquences (mots) *finis*, S^ω les mots *infinis* sur S.

$w = s_0 \dots s_n$ est un chemin de longueur n si $s_i \to s_{i+1}$ pour tout $0 \leq i < n$.

$\rho = s_0 s_1 \dots$ est un chemin infini si $s_i \to s_{i+1}$ pour tout $i \geq 0$.

Notation pour ST II

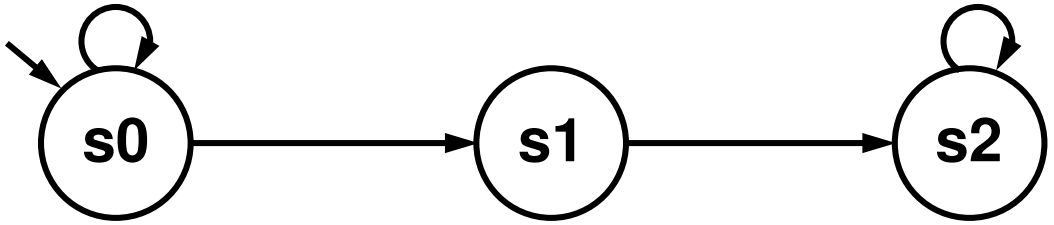
$\rho(i)$ dénote le i-ème élément de ρ et ρ^i le suffixe partant de $\rho(i)$.

$s \rightarrow^* t$ s'il existe un chemin de s à t.

$s \rightarrow^+ t$ s'il existe un tel chemin de longueur au moins 1.

Si $s \rightarrow^* t$ alors s est un predecesseur de t et t un successeur de s.

Exemple



$S = \{s_0, s_1, s_2\}$; racine s_0

$s_0 \to s_0$ $s_0 \to s_1$ $s_1 \to s_2$ $s_2 \to s_2$

$s_0 s_1 s_2$ est un chemin de longueur 2, $s_0 \to^* s_2$ et $s_0 \to^+ s_2$

$s_1 \to^* s_1$ mais $s_1 \not\to^+ s_1$

$\rho = s_0 s_0 s_1 s_2 s_2 s_2 \ldots$ est un chemin infini.

$\rho(2) = s_1$ $\rho^1 = s_0 s_1 s_2 s_2 s_2 \ldots$

ST finis et infinis

Plusieurs raisons font qu'un ST peut être infini:

- Données: entiers, réels, listes, arbres, pointeurs, ...
- Contrôle: récursion, création de threads dynamique ...
- Communication: canaux FIFO ...
- Paramètres: nombre de participants dans un protocôle ...
- Temps réel: continu ou discret

Certains (pas tout!) de ces caractéristics donnent lieu à des problèmes de vérification indécidables. Ici, on se concentrera sur les systèmes à états finis.

Structures de Kripke (SK)

Idée: Extraire des affectations de chaque état:

$$\mathcal{K} = (S, \rightarrow, r, AP, \nu)$$

$(S, \rightarrow, r)$	$\widehat{=}$	le ST sous-jacent
AP	$\widehat{=}$	ensemble de prédicats
$\nu : S \rightarrow 2^{AP}$	$\widehat{=}$	Interprétation des prédicats

Remarques:

- 2^{AP} dénote les *parties* de AP.
- On représente une affectation par le sous-ensemble des prédicats vrais.

Exemple d'une SK

ST $(S, \rightarrow, r)$ comme dans l'Exemple 1.

Supposons qu'on s'intéresse aux actions de production et consommation:

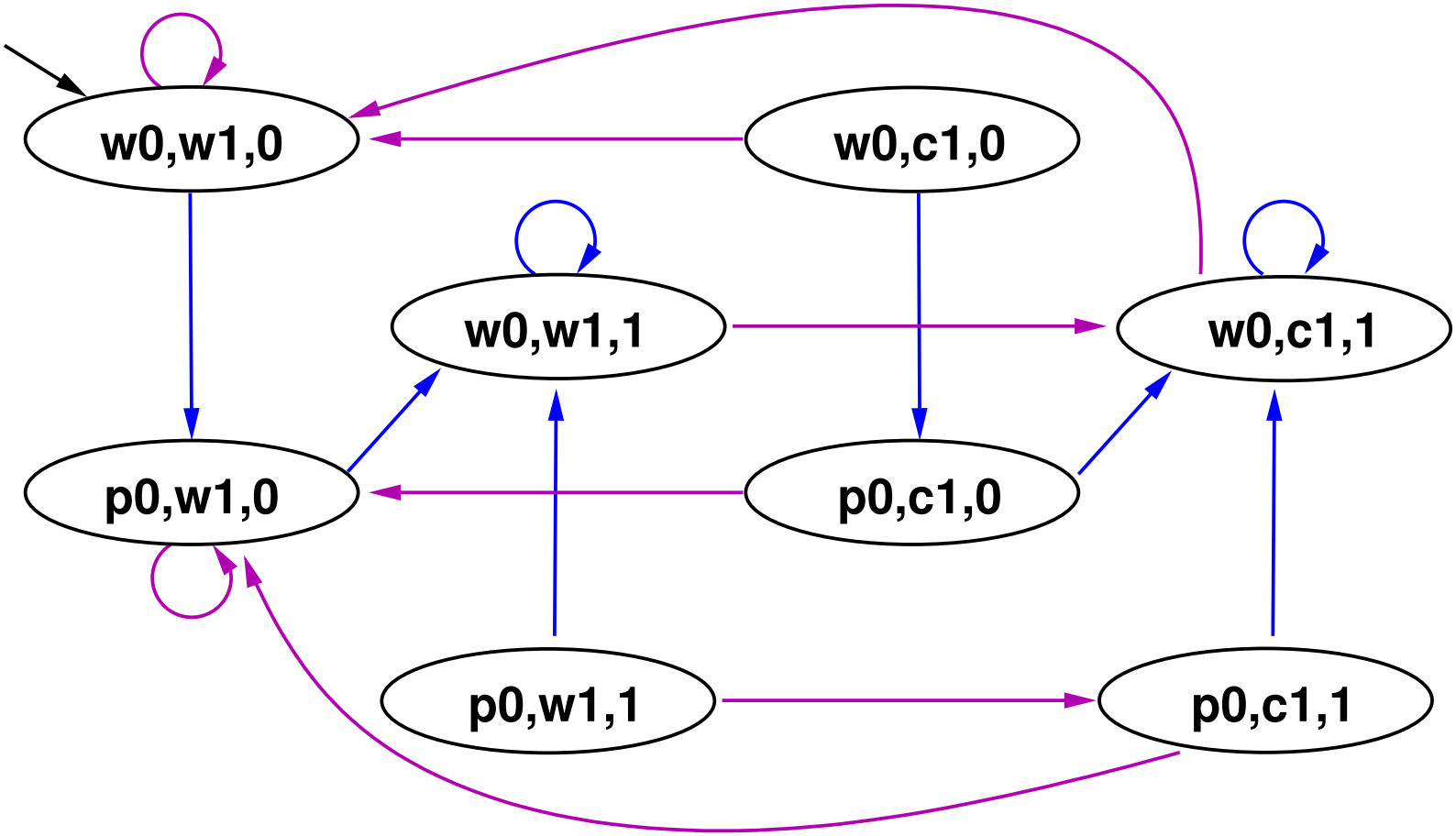
Soit $AP = \{prod, cons\}$;

$\nu^{-1}(prod) = \{p_0\} \times \{w_1, c_1\} \times \{0, 1\}$;

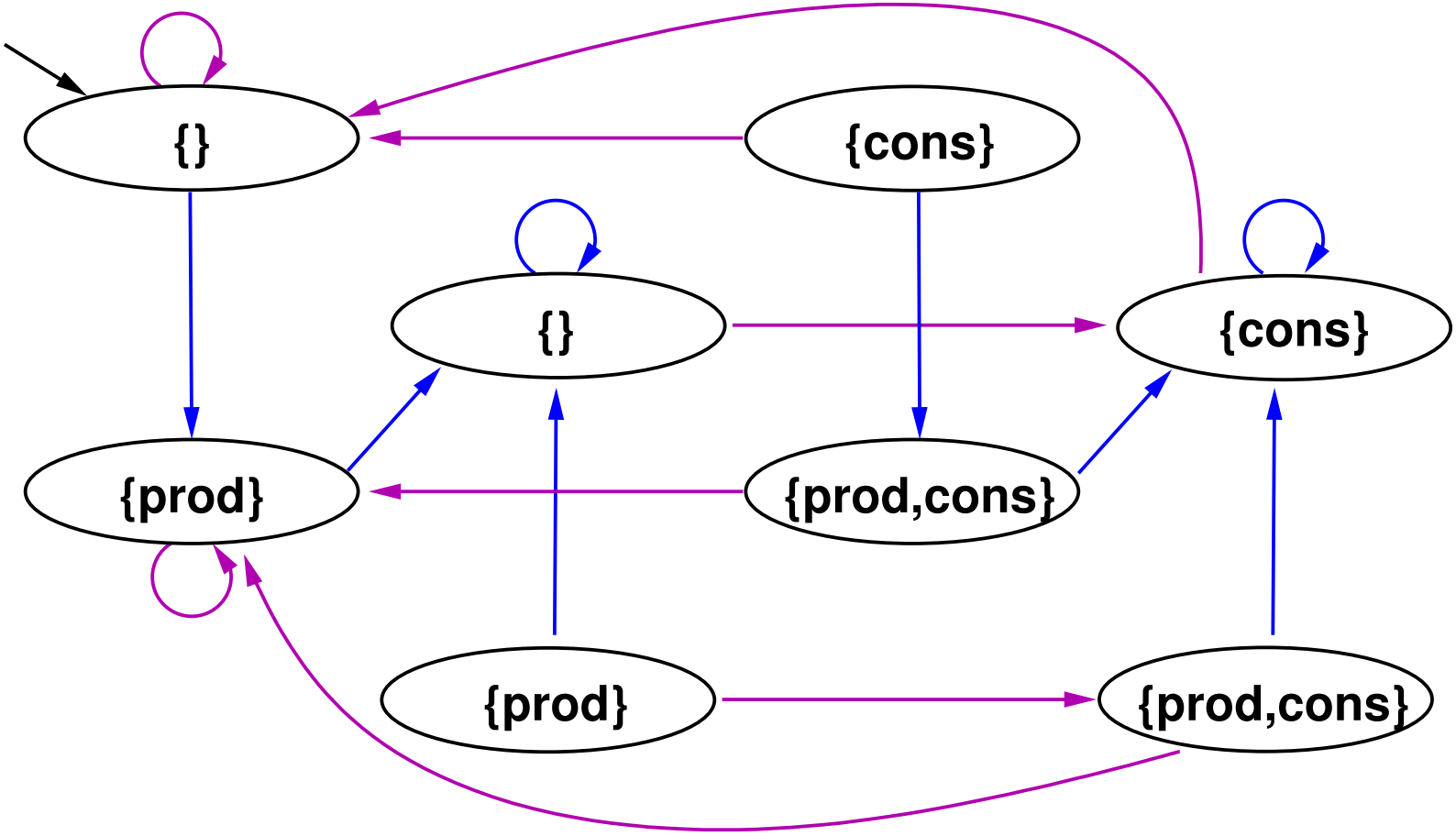
$\nu^{-1}(cons) = \{w_0, p_0\} \times \{c_1\} \times \{0, 1\}$.

Rappel: Exemple 1

Dans l'Exemple 1, . . .



. . . les affectations sont ainsi :

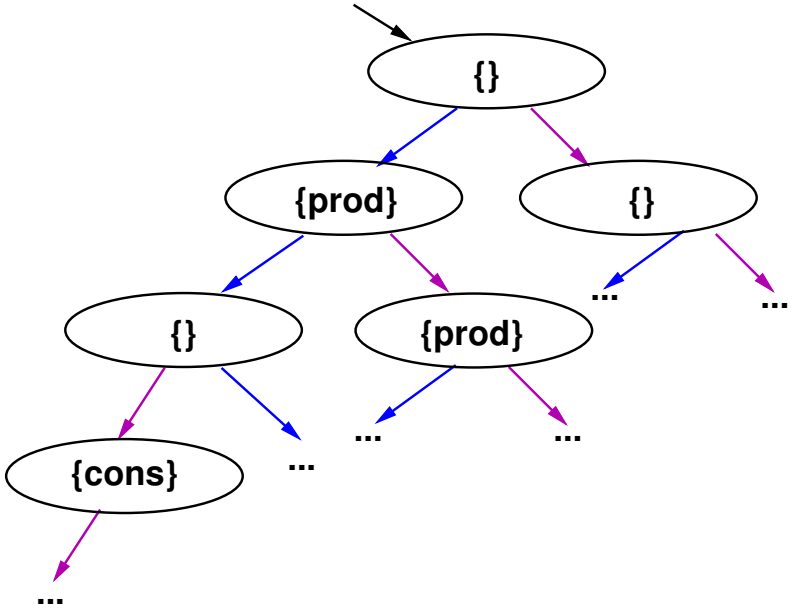


Séquences et arbres

Dans la logique linéaire, on considère les séquences :

p.ex. $\emptyset\ \emptyset\ \{prod\}\ \emptyset\ \{cons\}\ \ldots$ ou $\emptyset\ \{prod\}\ \{prod\}\ \{prod\}\ \ldots$

Dans la logique branchante on considère l'arbre des exécutions :



Exemples de propriétés

“*prod* et *cons* ne sont jamais vrais en même temps.”

(exemple d’une invariante)

“Après une production il peut y avoir une consommation.”

(exemple d’une propriété de vivacité)

Partie 3: Logique temporelle linéaire

Préliminaires

Idée: le temps progress de façon discrète et linéaire, chaque moment possède un seul successeur dans le futur

origines dans la philosophie et la logique

Exemple le plus prominent : LTL

- utilisé pour la vérification depuis les années 1970

Syntaxe de LTL

Soit AP un ensemble de prédicats. Les formules de LTL sur AP sont définies comme suit :

Si $p \in AP$ alors p est une formule.

Si ϕ_1, ϕ_2 sont des formules, alors aussi les suivants:

$$\neg\phi_1, \qquad \phi_1 \vee \phi_2, \qquad \mathbf{X}\,\phi_1, \qquad \phi_1\,\mathbf{U}\,\phi_2$$

Intuition: $\mathbf{X} \mathrel{\widehat{=}}$ “next” (prochain), $\mathbf{U} \mathrel{\widehat{=}}$ “until” (jusqu’à).

Remarques

C'est une syntaxe minimale qu'on utilisera pour des preuves.

Pour plus d'expressivité, on définit quelques raccourcis:

Comparaison de logique propositionnelle (LP) and LTL:

	LP	LTL
Syntaxe	prédicats, opérateurs logiques	+ modalités temporelles
Évaluée sur...	affectations	séquences d'affectations
Semantique	ensemble d'affectations	ensemble de séquences

Semantique de LTL

Soit ϕ une formule de LTL formula et σ une séquence d'affectations.
On écrit $\sigma \models \phi$ pour "σ satisfait ϕ."

$$
\begin{array}{lll}
\sigma \models & p & \text{if } p \in AP \text{ and } p \in \sigma(0) \\
\sigma \models & \neg\phi & \text{if } \sigma \not\models \phi \\
\sigma \models & \phi_1 \vee \phi_2 & \text{if } \sigma \models \phi_1 \text{ or } \sigma \models \phi_2 \\
\sigma \models & \mathbf{X}\,\phi & \text{if } \sigma^1 \models \phi \\
\sigma \models & \phi_1 \,\mathbf{U}\, \phi_2 & \text{if } \exists i : \left(\sigma^i \models \phi_2 \ \wedge \ \forall k < i : \sigma^k \models \phi_1\right)
\end{array}
$$

Semantique de ϕ: $\quad [\![\phi]\!] = \{\, \sigma \mid \sigma \models \phi \,\}$

Exemples

Soit $AP = \{p, q, r\}$. Trouver si la séquence

$$\sigma = \{p\}\ \{q\}\ \{p\}^{\omega}$$

satisfait les formules suivantes :

$$p$$

$$q$$

$$\mathbf{X}\, q$$

$$\mathbf{X}\, \neg p$$

$$p\, \mathbf{U}\, q$$

$$q\, \mathbf{U}\, p$$

$$(p \vee q)\, \mathbf{U}\, r$$

Raccourcis

On utilisera les définitions suivantes :

$$\begin{aligned}
\phi_1 \wedge \phi_2 &\equiv \neg(\neg\phi_1 \vee \neg\phi_2) & \qquad \mathbf{F}\,\phi &\equiv \mathbf{true}\,\mathbf{U}\,\phi \\
\phi_1 \rightarrow \phi_2 &\equiv \neg\phi_1 \vee \phi_2 & \qquad \mathbf{G}\,\phi &\equiv \neg\,\mathbf{F}\,\neg\phi \\
\mathbf{true} &\equiv a \vee \neg a & \qquad \phi_1\,\mathbf{W}\,\phi_2 &\equiv (\phi_1\,\mathbf{U}\,\phi_2) \vee \mathbf{G}\,\phi_1 \\
\mathbf{false} &\equiv \neg\mathbf{true} & \qquad \phi_1\,\mathbf{R}\,\phi_2 &\equiv \neg(\neg\phi_1\,\mathbf{U}\,\neg\phi_2)
\end{aligned}$$

Signification: $\mathbf{F} \mathrel{\widehat{=}}$ “finalement”, $\mathbf{G} \mathrel{\widehat{=}}$ “globalement” (toujours),
$\mathbf{W} \mathrel{\widehat{=}}$ “weak until”, $\mathbf{R} \mathrel{\widehat{=}}$ “release”.

Des exemples

Invariant: $\mathbf{G}\,\neg(cs_1 \wedge cs_2)$

cs_1 et cs_2 ne sont jamais vrais en même temps.

Sûreté: $(\neg x)\ \mathbf{W}\ y$

x n'apparaît pas avant y

Remarque: Si y n'apparaît jamais, alors x n'apparaît non plus.

Vivacité: $(\neg x)\ \mathbf{U}\ y$

x n'apparaît pas avant y et y apparaît sûrement.

Des exemples

$\mathbf{G}\,\mathbf{F}\,p$

p apparaît infiniment souvent.

$\mathbf{F}\,\mathbf{G}\,p$

À partir d'un moment, *p* tient toujours.

$\mathbf{G}(try_1 \rightarrow \mathbf{F}\,cs_1)$

Pour exclusion mutuelle: Si processus 1 essaie d'entrer dans la zone critique, il y parviendra.

Tautologie, équivalence

Tautologie: formule ϕ avec $[\![\phi]\!] = (2^{AP})^\omega$

Insatisfaisable: formule ϕ avec $[\![\phi]\!] = \emptyset$

Équivalence: formules ϕ_1, ϕ_2 avec iff $[\![\phi_1]\!] = [\![\phi_2]\!]$.
Notation: $\phi_1 \equiv \phi_2$

Équivalences: relations entre modalités

$$
\begin{aligned}
\mathbf{X}(\phi_1 \vee \phi_2) &\equiv \mathbf{X}\,\phi_1 \vee \mathbf{X}\,\phi_2 \\
\mathbf{X}(\phi_1 \wedge \phi_2) &\equiv \mathbf{X}\,\phi_1 \wedge \mathbf{X}\,\phi_2 \\
\mathbf{X}\,\neg\phi &\equiv \neg\,\mathbf{X}\,\phi \\
\\
\mathbf{F}(\phi_1 \vee \phi_2) &\equiv \mathbf{F}\,\phi_1 \vee \mathbf{F}\,\phi_2 \\
\neg\,\mathbf{F}\,\phi &\equiv \mathbf{G}\,\neg\phi \\
\\
\mathbf{G}(\phi_1 \wedge \phi_2) &\equiv \mathbf{G}\,\phi_1 \wedge \mathbf{G}\,\phi_2 \\
\neg\,\mathbf{G}\,\phi &\equiv \mathbf{F}\,\neg\phi \\
\\
(\phi_1 \wedge \phi_2)\,\mathbf{U}\,\psi &\equiv (\phi_1\,\mathbf{U}\,\psi) \wedge (\phi_2\,\mathbf{U}\,\psi) \\
\phi\,\mathbf{U}\,(\psi_1 \vee \psi_2) &\equiv (\phi\,\mathbf{U}\,\psi_1) \vee (\phi_1\,\mathbf{U}\,\psi_2)
\end{aligned}
$$

Équivalences: idempotence et recursion

$$\mathbf{F}\,\phi \equiv \mathbf{F}\,\mathbf{F}\,\phi$$

$$\mathbf{G}\,\phi \equiv \mathbf{G}\,\mathbf{G}\,\phi$$

$$\phi\,\mathbf{U}\,\psi \equiv \phi\,\mathbf{U}\,(\phi\,\mathbf{U}\,\psi)$$

$$\mathbf{F}\,\phi \equiv \phi \vee \mathbf{X}\,\mathbf{F}\,\phi$$

$$\mathbf{G}\,\phi \equiv \phi \wedge \mathbf{X}\,\mathbf{G}\,\phi$$

$$\phi\,\mathbf{U}\,\psi \equiv \psi \vee (\phi \wedge \mathbf{X}(\phi\,\mathbf{U}\,\psi))$$

$$\phi\,\mathbf{W}\,\psi \equiv \psi \vee (\phi \wedge \mathbf{X}(\phi\,\mathbf{W}\,\psi))$$

Interprétation de LTL sur une SK

Soit $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ une SK.
On s'intéresse aux séquence générées par $\mathcal{K}$.

Soit $\rho \in S^\omega$ un chemin infini dans $\mathcal{K}$.

On affecte à ρ son "image" $\nu(\rho)$ dans $(2^{AP})^\omega$; pour tout $i \geq 0$ soit

$$\nu(\rho)(i) = \nu(\rho(i))$$

alors $\nu(\rho)$ est la séquence d'affectations correspondante.

On note $[\![\mathcal{K}]\!]$ l'ensemble de ces séquences :

$$[\![\mathcal{K}]\!] = \{\, \nu(\rho) \mid \rho \text{ is an infinite path of } \mathcal{K} \,\}$$

Le problème de model-checking pour LTL

Problème: Étant donné une SK $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ et une formula de LTL ϕ sur AP, tester si $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$.

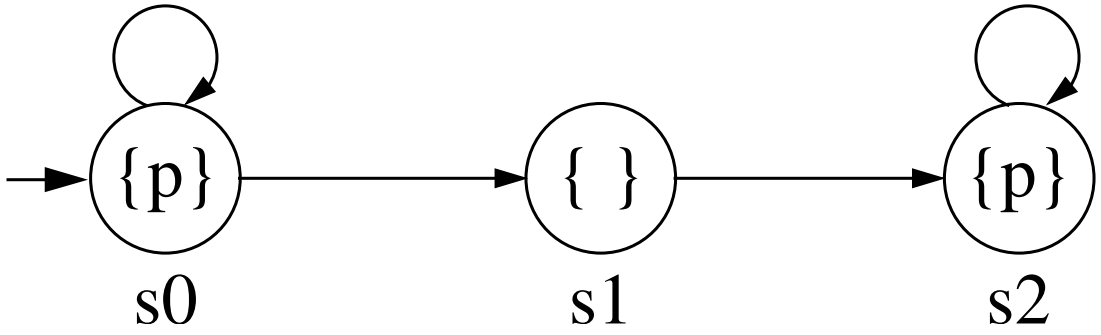
Définition: Si $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$ alors on écrit $\mathcal{K} \models \phi$.

Interprétation: Toute exécution de $\mathcal{K}$ satisfait ϕ.

Remarque: Il est possible que have $\mathcal{K} \not\models \phi$ et $\mathcal{K} \not\models \neg\phi$!

Exemple

On considère la SK suivante $\mathcal{K}$ avec $AP = \{p\}$:



Il y a deux espèces de chemins infinis dans $\mathcal{K}$:

(i) Soit le système reste dans s_0 à jamais,

(ii) soit il parvient à s_2 via s_1.

On a:

$\mathcal{K} \models \mathbf{F}\,\mathbf{G}\,p$

$\mathcal{K} \not\models \mathbf{G}\,p$

Partie 4: Automates de Büchi

Contenu

Problème de model-checking: $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$ – comment tester algorithmiquement?

On utilise la théorie des langages formels : $[\![\mathcal{K}]\!]$ et $[\![\phi]\!]$ sont des langages (de mots infinis).

Trouver une classe d'automates pour représenter ces langages.

Définir des opérations utiles sur ces automates pour résoudre le problème.

Automates de Büchi

Un automate de Büchi (AB) est un tuple $\mathcal{B} = (\Sigma, S, s_0, \Delta, F)$, où:

Σ	est un alphabet fini;
S	est un ensemble fini d'états;
$s_0 \in S$	est un état initial;
$\Delta \subseteq S \times \Sigma \times S$	sont des transitions;
$F \subseteq S$	sont des états acceptants.

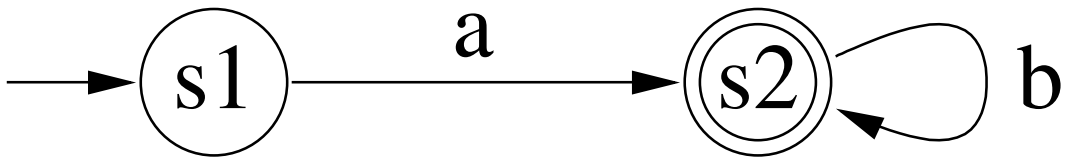
Remarques:

La définition et représentation graphique est identique aux automates finis.

Par contre, les AB travaillent sur les mots *infinis*, ce qui nécessite une condition d'acceptation différente.

Exemple

Répresentation d'un AB :



Les éléments de cet automate sont $(\Sigma, S, s_1, \Delta, F)$, avec:

- $\Sigma = \{a, b\}$ (symboles sur les arêtes)
- $S = \{s_1, s_2\}$ (cercles)
- s_1 (indiqué par une flèche entrante)
- $\Delta = \{(s_1, a, s_2), (s_2, b, s_2)\}$ (arêtes)
- $F = \{s_2\}$ (avec cercle double)

Langage d'un AB

Soit $\mathcal{B} = (\Sigma, S, s_0, \Delta, F)$ un automate de Büchi.

Un calcul de $\mathcal{B}$ sur un mot infini $\sigma \in \Sigma^\omega$ est une séquence infinie d'états $\rho \in S^\omega$ telle que $\rho(0) = s_0$ et $(\rho(i), \sigma(i), \rho(i+1)) \in \Delta$ pour tout $i \geq 0$.

On appelle ρ acceptant si $\rho(i) \in F$ pour un nombre infini de i.

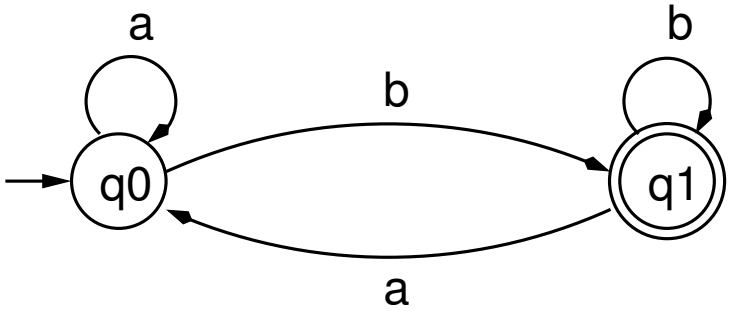
> Du coup, ρ contient infiniment souvent un état acceptant. Par le principe de tiroirs, il y a donc au moins un état acceptant qui est visité infiniment souvent.

$\sigma \in \Sigma^\omega$ est accepté par $\mathcal{B}$ s'il existe un calcul acceptant sur σ dans $\mathcal{B}$.

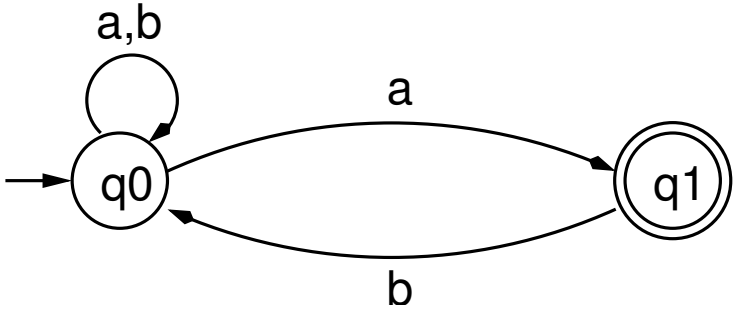
Le langage de $\mathcal{B}$, noté $\mathcal{L}(\mathcal{B})$, est l'ensemble de mots acceptés par $\mathcal{B}$.

Automates de Büchi: exemples

"infiniment souvent b"



"infiniment souvent ab"



Les automates de Büchi et LTL

Soit AP un ensemble de prédicats.

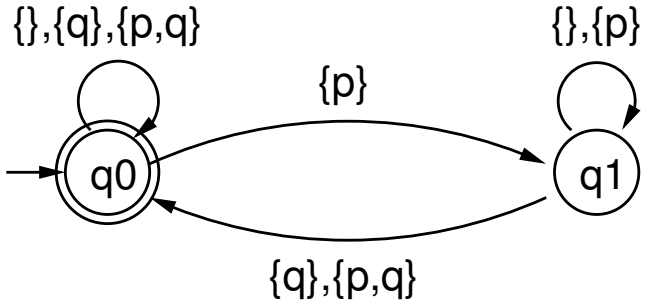
Un AB avec alphabet 2^{AP} accepte une séquence d'affectations.

Lemme: Pour toute formule de LTL il existe un AB $\mathcal{B}$ tel que $\mathcal{L}(\mathcal{B}) = [\![\phi]\!]$.

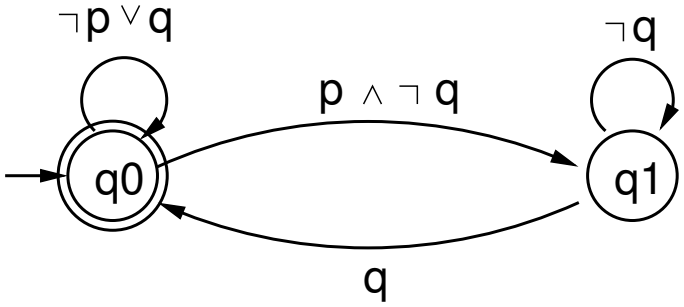
(On prouvera ce lemme plus tard.)

Exemples: $\mathbf{F}\, p$, $\mathbf{G}\, p$, $\mathbf{G}\,\mathbf{F}\, p$, $\mathbf{G}(p \rightarrow \mathbf{F}\, q)$, $\mathbf{F}\,\mathbf{G}\, p$

AB pour $\mathbf{G}(p \rightarrow \mathbf{F}\, q)$, avec $AP = \{p, q\}$.



On se permet d'étiqueter les arêtes avec des formules; dans ce cas, une formule F est un raccourci pour toutes les modèles de $[\![F]\!]$.



Opérations sur les AB

Les langages acceptés par des AB s'appellent ω-reguliers ou ω-reconnaissables.

Tout comme pour les langages reconnaissables sur mots finis, on a des propriétés de fermeture:

Si $\mathcal{L}_1$ et $\mathcal{L}_2$ sont ω-reconnaissables, alors les suivants le sont aussi :

$$\mathcal{L}_1 \cup \mathcal{L}_2, \qquad \mathcal{L}_1 \cap \mathcal{L}_2, \qquad \mathcal{L}_1^c.$$

On étudiera des opérations pour construire des AB pour certains cas.

Sur les transparents suivants on prend $\mathcal{B}_1 = (\Sigma, S, s_0, \Delta_1, F)$ et $\mathcal{B}_2 = (\Sigma, T, t_0, \Delta_2, G)$ (avec $S \cap T = \emptyset$).

Union

"Juxtaposer" $\mathcal{B}_1$ et $\mathcal{B}_2$, et ajouter un nouvel état initial.

Autrement dit, l'AB $\mathcal{B} = (\Sigma,\ S \cup T \cup \{u\},\ u,\ \Delta_1 \cup \Delta_2 \cup \Delta_u,\ F \cup G)$
accepte $\mathcal{L}(\mathcal{B}_1) \cup \mathcal{L}(\mathcal{B}_2)$, où

u est un état "frais" ($u \notin S \cup T$);

$\Delta_u = \{\, (u, \sigma, s) \mid (s_0, \sigma, s) \in \Delta_1 \,\} \cup \{\, (u, \sigma, t) \mid (t_0, \sigma, t) \in \Delta_2 \,\}$.

Intersection I (cas spécial)

On considère d'abord le cas ou tous les états de $\mathcal{B}_2$ acceptent ($G = T$).

Idée: Construire un automate de produit, tester si on visite F infiniment souvent.

Soit $\mathcal{B} = (\Sigma,\ S \times T,\ \langle s_0, t_0\rangle,\ \Delta,\ F \times T)$, avec

$$\Delta = \{\, (\langle s, t\rangle, a, \langle s', t'\rangle) \mid a \in \Sigma,\ (s, a, s') \in \Delta_1,\ (t, a, t') \in \Delta_2 \,\}.$$

Alors: $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$.

Intersection II (cas général)

Problème: La condition d'acceptance doit tester si *les deux* conditions d'acceptance sont satisfaites, potentiellement à des instants différents.

Idée: construire deux copies de l'automate de produit.

- La première copie attend un état de F.
- La seconde copie attend un état de G.
- Ayant trouvé un tel état, on bascule entre les deux copies.
- La condition d'acceptance assure qu'on bascule infiniment souvent.

Soit $\mathcal{B} = (\Sigma, U, u, \Delta, H)$, où

$$U = S \times T \times \{1,2\}, \quad u = \langle s_0, t_0, 1\rangle, \quad H = F \times T \times \{1\}$$

$$(\langle s,t,1\rangle, a, \langle s',t',1\rangle) \in \Delta \text{ iff } (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ s \notin F$$
$$(\langle s,t,1\rangle, a, \langle s',t',2\rangle) \in \Delta \text{ iff } (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ s \in F$$
$$(\langle s,t,2\rangle, a, \langle s',t',2\rangle) \in \Delta \text{ iff } (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ t \notin G$$
$$(\langle s,t,2\rangle, a, \langle s',t',1\rangle) \in \Delta \text{ iff } (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ t \in G$$

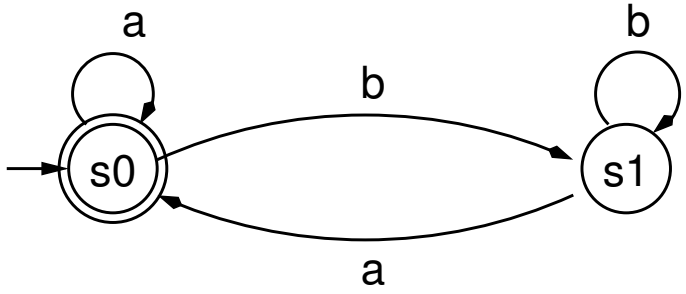
Remarques:

On démarre dans la première copie.

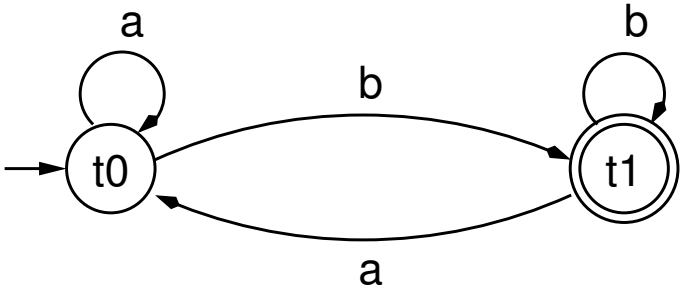
$S \times G \times \{2\}$ marche aussi comme condition d'acceptane.

On peut généraliser la construction à l'intersection entre n automates.

Intersection: exemple

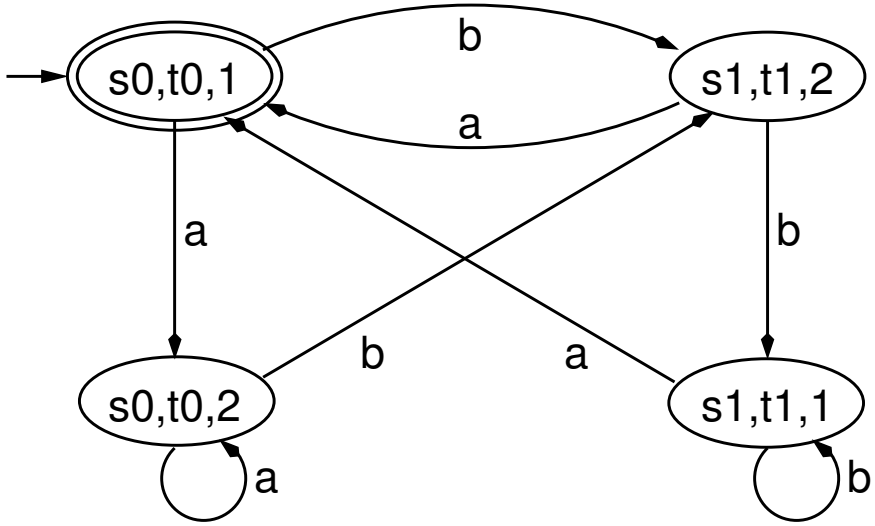


B1



B2

B1 x B2



Complément

Problème: Étant donné $\mathcal{B}_1$, construire $\mathcal{B}$ avec $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}_1)^c$.

Une telle construction est possible (mais compliqué). On n'en a pas besoin pour ce cours.

Littérature:

Wolfgang Thomas, *Automata on Infinite Objects*,
Chapter 4 in *Handbook of Theoretical Computer Science*

Igor Walukiewicz, lecture notes on *Automata and Logic*, chapter 3,
`www.labri.fr/perso/igw/Papers/igw-eefss01.pdf`

Grädel, Thomas, Wilke (eds.), *Automata, Logics, and Infinite Games*,
LNCS 2500, chapîtres 3 et 4.

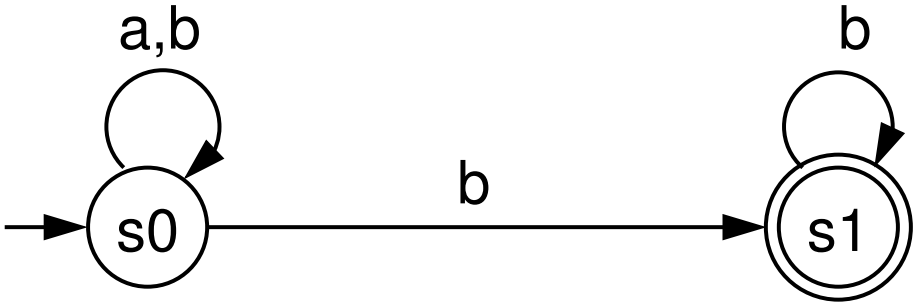
AB déterministes

Dans les cas des automates finis, il est bien connu que les automates finis *déterministes* sont aussi expressifs que les non-déterministes.

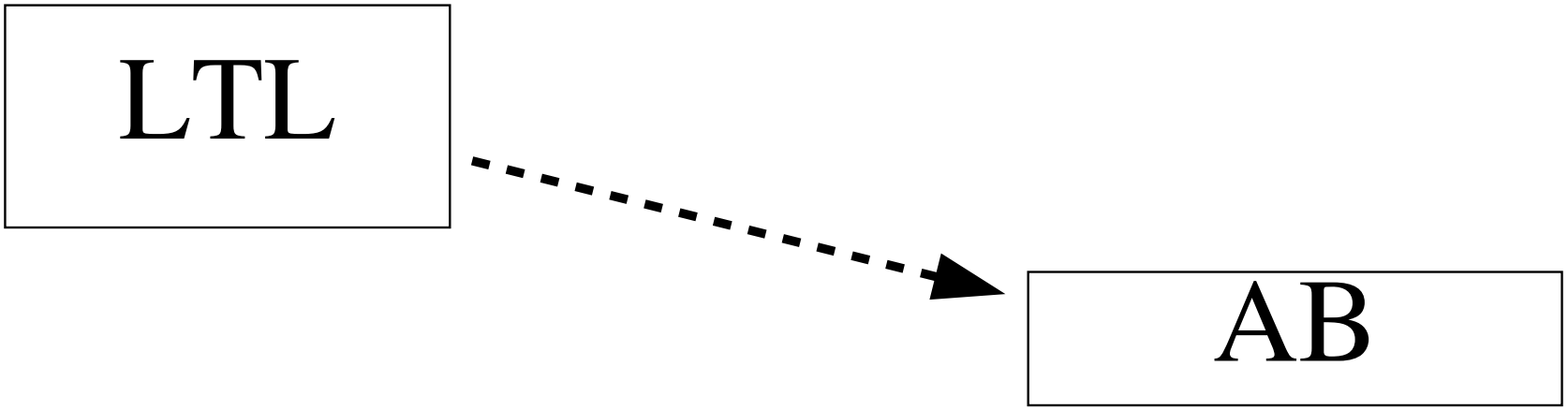
Ceci n'est pas vrai pour les automates de Büchi.

Contre-exemple : il n'y a pas d'AB déterministe équivalent à l'AB ci-dessous:

"Nombre fini de *a*."

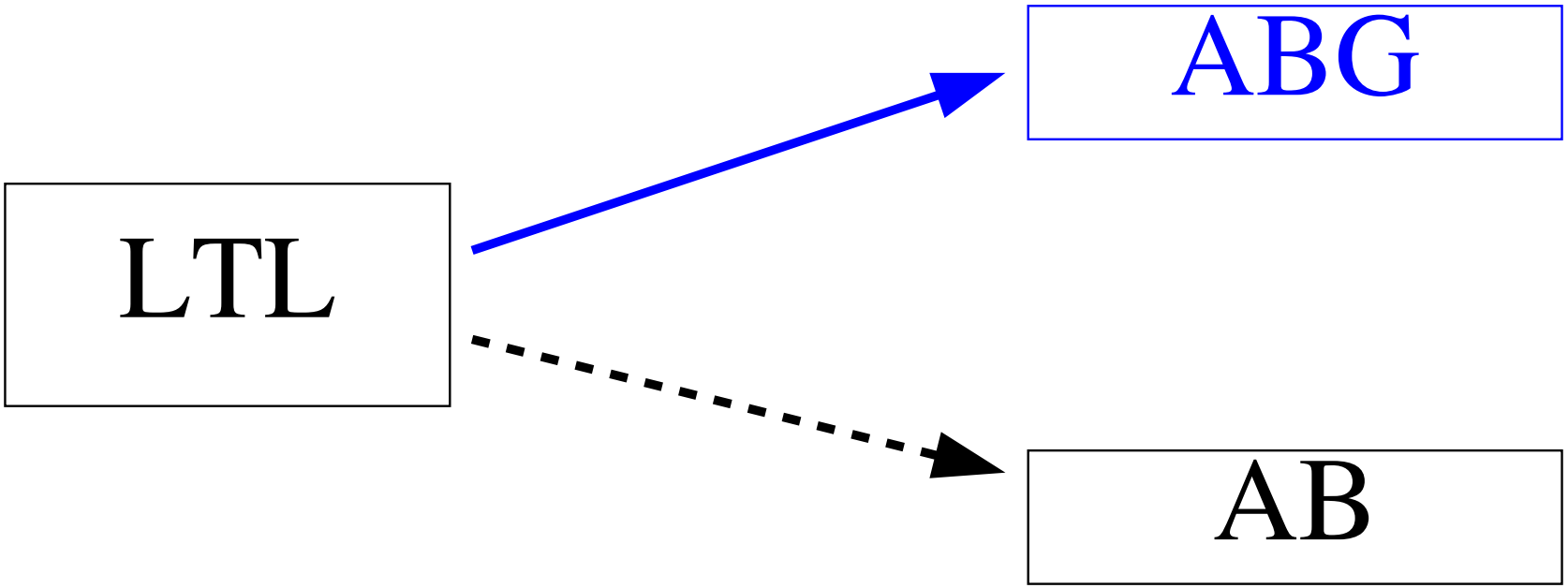


Aperçu



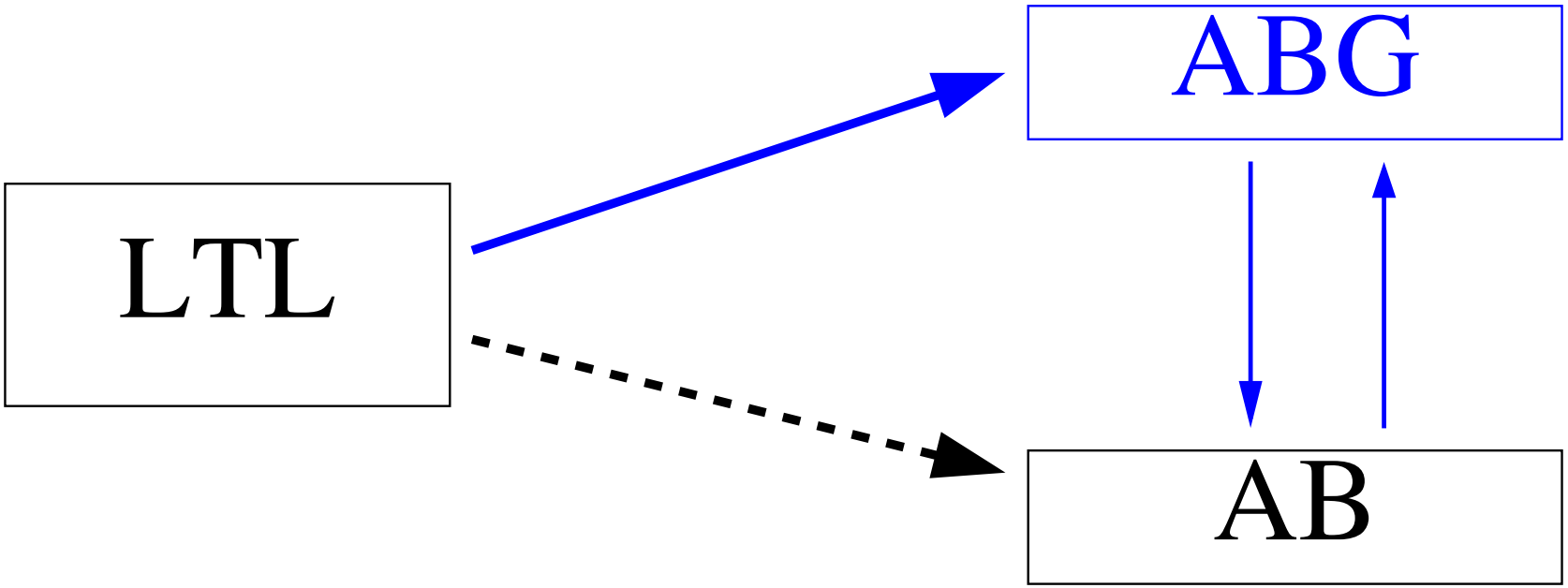
Nous souhaitons traduire les formules de LTL en automate de Büchi.

Aperçu



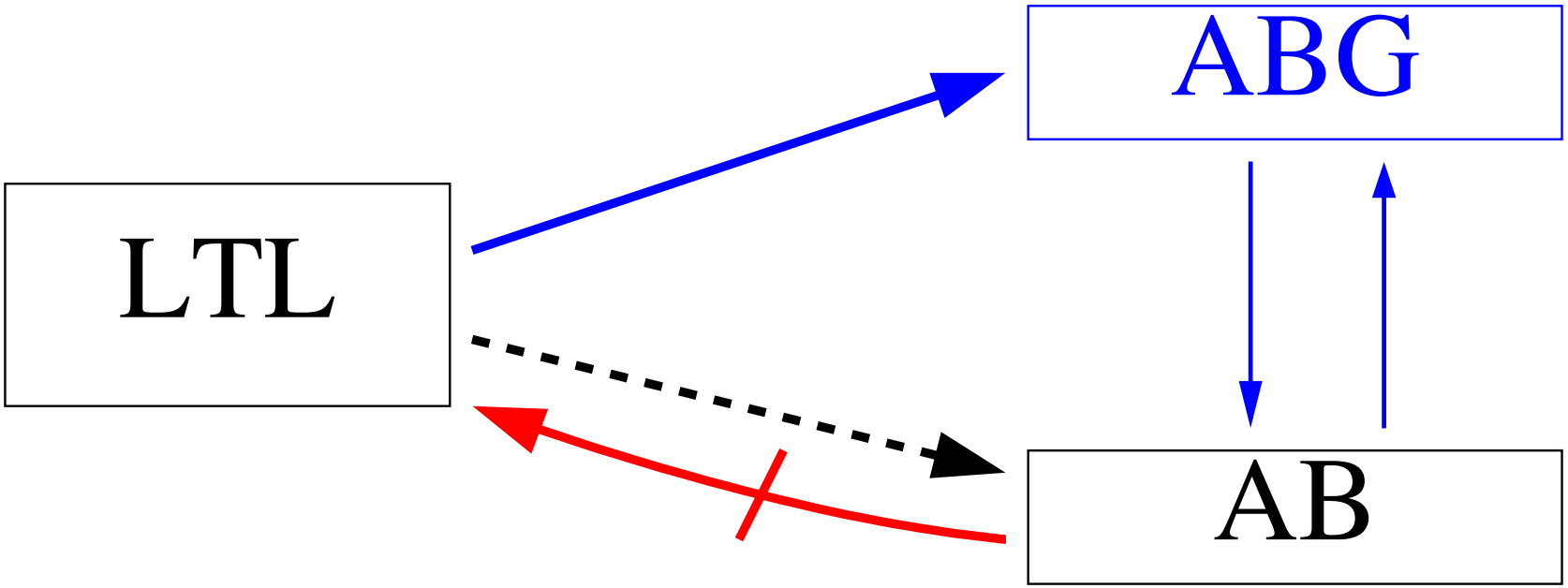
Détour: On va les traduire en *automates de Büchi généralisés* (ABG).

Aperçu



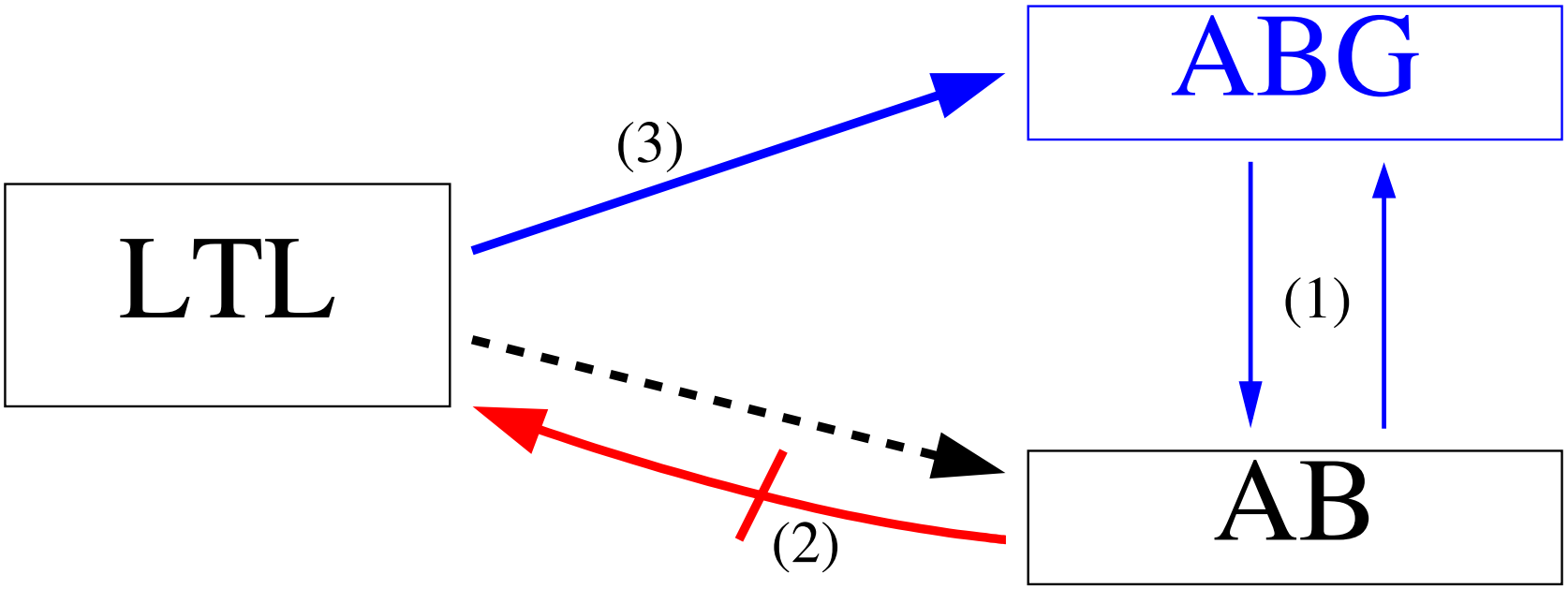
Les ABG accepteront les même langages que les AB.

Aperçu



La traduction inverse (AB $\rightarrow$ LTL) n'est pas possible en général.

Aperçu



On procède dans l'ordre indiqué.

(1) AB généralisés

Un automate de Büchi généralisé (ABG) est un tuple $\mathcal{G} = (\Sigma, S, s_0, \Delta, \mathcal{F})$.

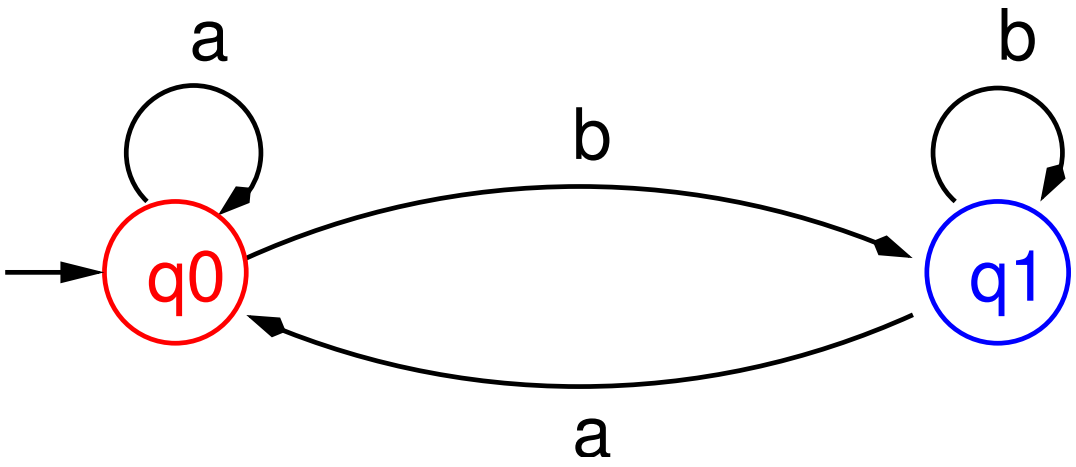
Il y a une seule différence par rapport aux AB normaux:

La condition d'acceptance $\mathcal{F} \subseteq 2^S$ est un *sous-ensemble des parties de S*.

P.ex., soit $\mathcal{F} = \{F_1, \ldots, F_n\}$. Un calcul ρ de $\mathcal{G}$ est acceptant si pour tout F_i ($i = 1, \ldots, n$), ρ contient un nombre infini d'occurrences d'états dans F_i.

ABG: Exemple

Pour le ABG ci-dessous, soit $\mathcal{F} = \{ \{q_0\}, \{q_1\} \}$.



Langage de l'automate: "infiniment souvent *a et* infiniment souvent *b*"

Note: Les ensembles de $\mathcal{F}$ ne sont pas forcement disjoints.

Avantage: Les ABG sont plus succints que les AB.

Traduction AB $\leftrightarrow$ ABG

Les ABG acceptent les mêmes langages que les AB.

AB $\rightarrow$ ABG: (trivial)

Soit $\mathcal{B} = (\Sigma, S, s_0, \Delta, F)$ un AB.

Alors $\mathcal{G} = (\Sigma, S, s_0, \Delta, \{F\})$ est un ABG tel que $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{B})$.

ABG $\rightarrow$ AB:

Soit $\mathcal{G} = (\Sigma, S, s_0, \Delta, \{F_1, \ldots, F_n\})$ un ABG.

On construit $\mathcal{B} = (\Sigma, S', s'_0, \Delta', F)$ comme suit:

$S' = S \times \{1, \ldots, n\}$

$s'_0 = (s_0, 1)$

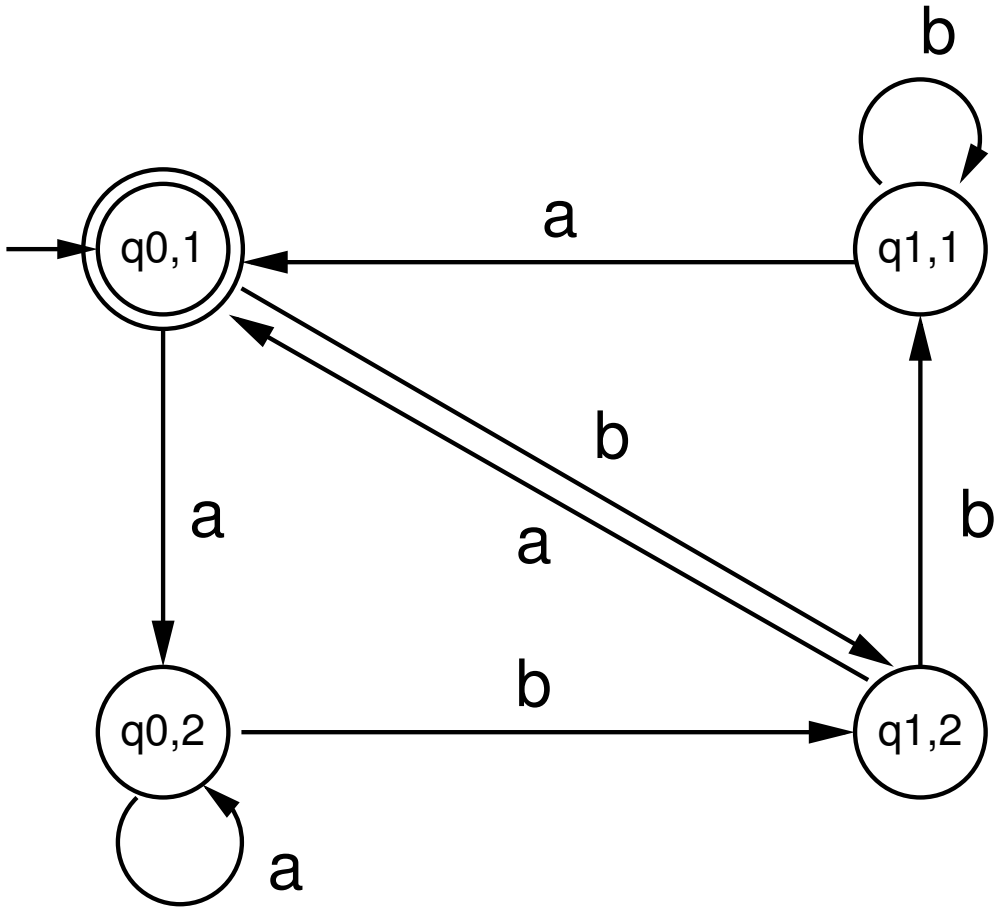
$F = F_1 \times \{1\}$

$((s, i), a, (s', k)) \in \Delta'$ ssi $1 \leq i \leq n,\ (s, a, s') \in \Delta$
et $k = \begin{cases} i & \text{si } s \notin F_i \\ (i \bmod n) + 1 & \text{si } s \in F_i \end{cases}$

Alors $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{G})$. (Idée: intersection spécialisée de n automates)

ABG $\rightarrow$ AB: exemple

L'AB correspondant à l'ABG précédent ("infiniment souvent *a et* infiniment souvent *b*") est comme suit:



Remarque: États initiaux multiples

Les définitions données pour les AB(G) contiennent exactement un état initial.

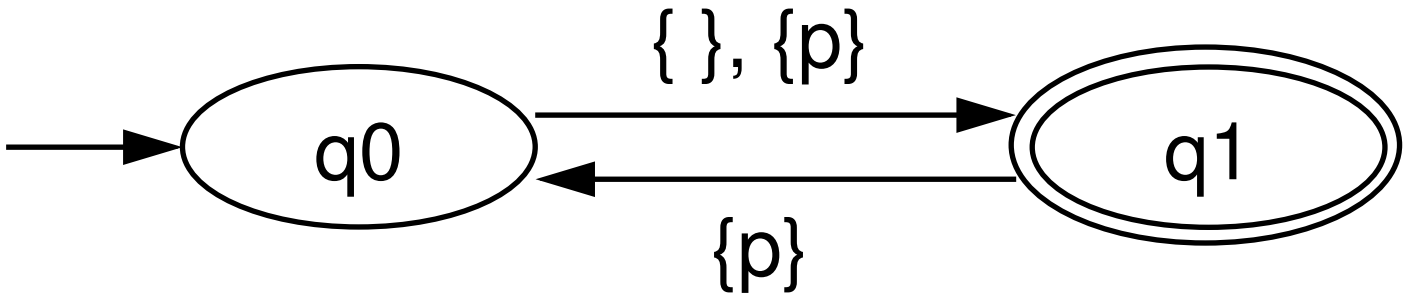
Pour la traduction LTL $\rightarrow$ ABG il conviendra d'avoir de multiples états initiaux (où un mot est accepté s'il est accepté à partir de n'importe quel état initial).

Évidemment, cette extension ne change pas le pouvoir d'expression des AB(G).

(2) Traduction AB $\rightarrow$ LTL

Cette tradition n'est pas possible en général.

Il existe des AB $\mathcal{B}$ tels qu'aucune formule de LTL ϕ ne satisfait $\mathcal{L}(\mathcal{B}) = [\![\phi]\!]$ (Wolper, 1983).



La propriété "*p* tient dans tous les deux coups" n'est pas expressible en LTL.

Preuve (AB $\not\to$ LTL)

On va d'abord montrer un lemme différent:

> Soit ϕ une formule LTL quelconque sur AP et n le nombre d'opérateurs $\mathbf{X}$ dans ϕ. On considère les séquences
>
> $$\sigma_i = \{p\}^i \emptyset \{p\}^\omega$$
>
> pour $i \geq 0$. Pour toute paire $i, k > n$ on a : $\sigma_i \models \phi$ iff $\sigma_k \models \phi$.

Preuve par récurrence sur ϕ:

> Si $\phi = p$, où $p \in AP$, alors $n = 0$ et $i, k \geq 1$.
> Du coup, $\sigma_i \models p$ et $\sigma_k \models p$.
>
> Pour les autres cas, on suppose que la propriété est vraie pour ϕ_1 und ϕ_2, c.à.d. si ϕ_1, ϕ_2 contiennent n_1 et n_2 occurrences de $\mathbf{X}$, respectivement, alors pour tout $i_1, k_1 > n_1$ on a $\sigma_{i_1} \models \phi_j$ ssi $\sigma_{k_1} \models \phi_j$, pour $j = 1, 2$.

Si $\phi = \neg\phi_1$, alors la preuve est immédiate.

Pour $\phi = \phi_1 \vee \phi_2$: pareil

Si $\phi = \mathbf{X}\,\phi_1$, alors $n_1 = n - 1$. Comme $i - 1, k - 1 > n - 1 = n_1$, l'hypothèse de récurrence implique: $\sigma_i^1 = \sigma_{i-1} \models \phi_1$ ssi $\sigma_k^1 = \sigma_{k-1} \models \phi_1$, ce qui conclut la preuve.

Pour $\phi = \phi_1\ \mathbf{U}\ \phi_2$: Soit $m > n$. On a:

$$\phi_1\ \mathbf{U}\ \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{X}(\phi_1\ \mathbf{U}\ \phi_2))$$

En itérant cette transformation on obtient:

$$\sigma_m \models \phi \quad \text{ssi} \quad \sigma_m \models \phi_2 \vee (\sigma_m \models \phi_1 \wedge (\sigma_{m-1} \models \phi_2 \vee (\ldots$$
$$(\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1\ \mathbf{U}\ \phi_2))))$$

Selon l'hypothèse de récurrence, on peut remplacer les indices plus grands que n par $n+1$ sans rien changer.

$$\sigma_m \models \phi \quad \text{ssi} \quad \sigma_{n+1} \models \phi_2 \vee (\sigma_{n+1} \models \phi_1 \wedge (\sigma_{n+1} \models \phi_2 \vee (\ldots (\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1 \,\mathbf{U}\, \phi_2))))$$

Ceci se simplifie comme suit:

$$\sigma_m \models \phi \quad \text{ssi} \quad \sigma_{n+1} \models \phi_2 \vee (\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1 \,\mathbf{U}\, \phi_2)$$

Du coup, la validité de $\sigma_m \models \phi$ est entièrement indépendante de m, ce qui implique la propriété souhaitée pour i et k.

Supposons maintenant qu'il existe une formule LTL ϕ qui exprime le langage de l'AB précédent ("*p* tient dans tous les deux coups"). Soit n le nombre de $\mathbf{X}$ dans ϕ.

On considère les séquences σ_{n+1} et σ_{n+2}. Si n est pair alors $\sigma_{n+1} \not\models \phi$ et $\sigma_{n+2} \models \phi$. Si n est impair, c'est l'inverse.

Pourtant, le lemme précédent nous dit que ceci est impossible: soit σ_{n+1} et σ_{n+2} satisfont ϕ tous les deux, soit ni l'une ni l'autre. Du coup, une telle formule ϕ n'existe pas.

Resoudre le problème de model-checking pour LTL

Étant donné une SK $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ et une formule de LTL ϕ sur AP, on demande si $\mathcal{K} \models \phi$.

Solution:

On interprète $\mathcal{K}$ comme un AB $\mathcal{B}_\mathcal{K}$:

$\mathcal{B}_\mathcal{K} = (2^{AP}, S, r, \Delta, S)$, où $\Delta = \{ (s, \nu(s), t) \mid s \rightarrow t \}$

Évidemment, $[\![\mathcal{K}]\!] = \mathcal{L}(\mathcal{B}_\mathcal{K})$.

On traduit $\neg\phi$ en un AB (généralisé) $\mathcal{B}_{\neg\phi}$.

On a alors:

$$
\begin{array}{rrcl}
 & \mathcal{K} & \models & \phi \\
\iff & [\![\mathcal{K}]\!] & \subseteq & [\![\phi]\!] \\
\iff & [\![\mathcal{K}]\!] & \cap & [\![\neg\phi]\!] = \emptyset \\
\iff & \mathcal{L}(\mathcal{B}_{\mathcal{K}}) & \cap & \mathcal{L}(\mathcal{B}_{\neg\phi}) = \emptyset
\end{array}
$$

Du coup:

On construit les AB $\mathcal{B}_{\mathcal{K}}$ et $\mathcal{B}_{\neg\phi}$.

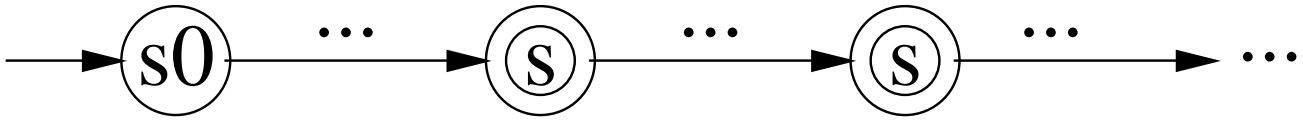
On les intersecte en utilisant le cas spécial car tous les états de $\mathcal{B}_{\mathcal{K}}$ sont acceptants.

Le problème de model-checking se reduit alors au problème du vide pour cette intersection.

Problème du vide

Problème: Étant donné un AB $\mathcal{B}$, tester si $\mathcal{L}(\mathcal{B}) = \emptyset$.

Observation: $\mathcal{L}(\mathcal{B}) \neq \emptyset$ ssi il existe $s \in F$ tel que $s_0 \rightarrow^* s \rightarrow^+ s$.



Solution efficace:

Identifier les composantes fortement connexes de $\mathcal{B}$ accessibles depuis s_0.

Tester s'il en existe une qui est *non triviale* (contient au moins une transition) et qui contient un état acceptant.

Algorithme de Tarjan: $\mathcal{O}(|\mathcal{B}|)$ temps et espace